2023 - CS603 - Summer - Sebastian Mohr - 23141808

1

a

1. class methods need an invariant, that holds before and after all methods execution, regardless of if the method is private or public.

```
class x {
  int y := 0;

  predicate P() {
    0 <= y < 20
  }

  method addToY(z: int)
    requires 0 <= z < 20
    requires P()
    ensures P()
  {
    if(y + z < 20 && y + z >= 0)
      y := y + z;
  }
}
```

2. object constructors need to make sure, that the invariant holds after the constructor execution and have preconditions for each input variable.

```
class x {
  int y;

  predicate P() {
    0 <= y < 20
  }

  constructor(y: int)
    requires 0 <= y < 20
    ensures P()
  {
    this.y := y
  }
}
```

3. Overridden methods must have equal or weaker preconditions and equal or stronger postconditions as the super method. Also the superclasses invariant must hold after

the method execution.

```
class x {
  /*...*/
  method addNumbers(y: int, z:int) returns (a: int)
    requires 0 < y < 10
    requires 0 < z < 10
    ensures a == y + z
    ensures Valid()
  {
    a := y + z;
  }
}

class x2 extends x {
  override method addNumbers(y: int, z:int) returns (a: int)
    requires -10 < y < 10
    requires -10 < z < 10
    ensures y + z > 0 :: a == y + z
    ensures y + z <= 0 :: a == 0
    ensures Valid()
  {
    if (y+z > 0) {
      a := y + z;
    } else {
      a := 0
    }
  }
}
```

b

```
{X >= 0 && Y >= 0} // pre
{0 == X * 0}
  {0 == 0}
int i = 0;

{0 == X * i }

int z = 0;
{z == X * i }

while(i != Y) {
  {i != Y} // guard
  {z == X * i} // invariant
  i = i + 1;

  {z == X * (i - 1)} // invariant
    {z == X * i - X} // simplification

  z = z + X;
  {z == X * i} // invariant
}
{i == Y} // !guard
{z == X * i && i == Y} // invariant & !guard
{z == X * Y} // post
```

c

The main difference between proving partial and total correctness is, that for total correctness the termination of a program has to be proven alongside its functionality.

Usually in total correctness prove there has to be proven, that a loop is always terminating at some point. For the example in (b) there has to be an extra loop invariant, that proves that $Y - i$ is decreasing in each iteration.

In (b) you could use an additional temporary variable, that stores the previous value of $Y - i$ in each loop iteration. This value could be compared to the current value of $Y - i$ and must always be bigger in each iteration, until at some point $Y - i$ becomes 0 and the loop gets broken.

d

Under-specification means that a program is not fully specified, and that there are situations that could occure that are not covered by the specification. In the current example of array sorting it could mean, that the specification doesn't state what happens with values that are duplicated.

```
method sortArray(a: Array<int>)
  requires a.length > 0
  ensures forall x :: 0 <= x < a.length - 1 || a[x] <= a[x+1]
{
  for(int i = 0; i < a.length - 1; i++) {
    for(int j = 0; j < (a.length - i - 1); j++) {
      if (a[j] > a[j+1]) {
        a[j], a[j+1] = a[j+1], a[j];
      }
    }
  }
}
```

This algorithm is not under specified, as it does state that all elements are smaller or equal to the next one, which means if there are duplicates it just orders them after each other.

2

a

I

```
ghost function Sum0(lo: int, hi: int): int
  requires lo <= hi
  decreases hi - lo
{
  ...
}

ghost function Sum1(lo: int, hi: int): int
  requires lo <= hi
  decreases hi - lo
{
  ...
}
```

II

```
lemma Sum2(lo: init, hi: int)
  requires lo <= hi
  decreases hi - lo
  ensures lo != hi ==> F(lo) + Sum=(lo + 1, hi) == Sum1(lo, hi)
{
  ...
}
```

III

Ghost code can help during the verification, but is treated as private functions.

IV

Lemmas help the verifier by assuming different mathematical expressions and thus reminding the verifier. They don't manipulate data.

b

I

```
assert a.Length > 0
while (i < a.Length)
  // loop variant
  decreases a.Length - i
  // loop invariants
  invariant 0 <= i <= a.Length
  invariant forall j || 0 <= j < i :: a[j] >= min
  invariant exists j || 0 <= j < i :: a[j] == min
{
  ...
}
```

The loop **invariant** is checked at the beggining and end of every loop iteration. The verifier needs it to prove that the expected postcondition will hold after the loop has terminated.

The loop **variant** is used to make sure that the loop terminates and prove total correctness.

II

The verification error would be that the second invariant (exists j ...) wouldn't hold anymore, as the verifier can't prove that 0 is a value stored in the array.

III

The verification error would be that the array index is out of bounds, as the array is sure to be of length 0 and therefore has no values stored. When trying to access the element `a[0]`, the verifier notices that this element is not accessible.

c

```
class Queue {
  var q : Seq<int>

  invariant
    forall i, j | 0 <= i <= j < |q| :: q[i] != q[j]

  constructor()
    ensures q == []
  {
    q := [];
  }

  method push(x: int)
    modifies q
    requries forall i | 0 <= i < |q| :: q[i] != x
    ensures old(|q|) == |q| - 1
    ensures forall i | 0 <= i < old(|q|) :: q[i] == old(q[i])
    ensures q[|q| - 1] == x
  {
    q := q + [x];
  }

  method pop() returns (x: int)
    modifies q
    requires |q| > 0
    ensures x == old(q[|q| - 1])
    ensures old(|q|) == |q| + 1
    ensures all i | 0 <= i < |q| :: q[i] == old(q[i])
  {
    x := q[|q| - 1];
    q := q[0..(|q|-2)];
  }
}
```

The class invariant must be true before and after each method. It states that each element is unique.

`push()` requires the new value to be unique in the queue. Also it ensures that all the elements in the queue stay the same, except for the last element.

`pop()` requires no precondtions from the client. For example, the `pop()` function only takes the last element and leaves the remaining elements untouched. Also the length only decreases by 1 with each method call. The return value is always the last element in the queue.

The constructor ensures that the client gets a fresh `Queue` without any elements stored. Also it ensures that the invariant holds at the class initialization.

4

a

Specification & System Model -> Promela file -> spin -> pan.c -> gcc -> pan -> execute -> error (trail file) or no errors = model holds

- **system model**: that's the code that gets executed
- **formula to be checked**: LTL formula that holds at some point
- **invariant properties**: never claims, LTL formulas that hold always
- **labels**: different code parts can be labeled to check if they get accessed at some point

```
bit x = 0;
bit y = 0;

active proctype P { // model
  do:
    x = x + 1;
    assert(x == 1);
    critSection: // label
      y = y + 1;
      assert(y == 1);
    x = 0;
    y = 0;
  od;
}

ltl A { <> x == 1 } // formula
ltl B { <> P@critSection } // label
ltl C { [] (x <= 1 && y <= 1)} // invariant
ltl D { [] (x == 1 => <> (y == 1))}
```

b

1. **G (coffeeLow => N alarmRings):** When the coffee is low, then the nextState has to ring the alarm.
2. **F (goodSalary && enjoyJob):** At some point in the future goodSalary and enjoyJob will both be true.
3. **G F (doorLocked U doorUnlocked):** Globally when the door is locked sometimes in the future, it's locked until it is unlocked.
4. **G ((shopOpen F frontOfQueueServed) => F queueIsServed U endOfQueueIsReached):** Globally when the shop is open and sometimes in the future the front of the queue is served, the whole queue will be served eventually until the end of the queue is reached.

c

LTL only looks at the possibility of certain events occuring during the runtime of the program. CTL also defines how often something occures, by defining if it's at least a single path through a program or every path.

1. Safety properties: Safety properties have to determine, that "something bad doesn't happen". This can achived by creating never claims, usually through LTL-formulas. They can then say something like: [] (x <= 1), which would mean that x can never be greater than 1.
2. Liveness properties: Liveness properties state, that "something good will happen". This can be achived by creating claims, that state that at one point a variable will have a specific value. For example the CTL formula: <> (y == 1), which would mean that at some point during program execution y will be 1.
3. Fairness properties: Fairness properties state, that every request will at some point be fulfilled. This can be achived by having labels in the program, that have to be visited at some point. For example: [] (a == 1 => <> b == 1), which means that if a == 1 (the request), then at some point b has to be 1 (the response).